

Ceramic Tiling Viewer
Honours Project

Landery Burt

1 November 1996

Abstract

Observing and interacting with multi-thread programs can be difficult for the programmer. Simple input/output (I/O) can become a nightmare when multiple threads read and write simultaneously. A solution would separate the I/O streams of the multiple threads, windowing techniques can achieve this.

This honours project report presents the design and implementation of Ceramic, a development tool which assists in observing and interacting with multi-thread programs. Multiple viewers (windows) can be opened to control I/O streams of multiple threads. Ceramic has an object-oriented design based on design patterns captured from Mössenböck's Oberon0 viewer system. Another feature are the hierarchical tiling viewers which are a hybrid of Elastic Windows developed by Kandogan & Shneiderman. Tiling viewers have some significant advantages over overlapping windows which Ceramic has exploited.

Contents

1	Introduction	2
1.1	Context	3
1.2	Problem	3
1.3	Solution	3
1.4	Example	3
1.5	Report Structure	4
2	Background & Motivation	6
2.1	Tiling Window Systems	6
2.1.1	Cedar	7
2.1.2	Oberon	7
2.1.3	Oberon0	8
2.1.4	CUBRICON	8
2.1.5	Elastic Windows	9
2.2	Design Patterns	9
2.3	Motivation	10
3	Project Objectives	12
4	Oberon0 Viewer System	13
4.1	Oberon0 Pattern	13
4.2	Pattern 1: Viewer	16
4.3	Pattern 2: Text Editor	18
5	Ceramic Tiling Viewer	21
5.1	Design Issues	21
5.1.1	Hierarchical Tiling Viewers	21
5.1.2	Virtual Port	21
5.1.3	Qt Toolkit	22
5.2	User Interface	22
5.2.1	Server	22
5.2.2	Client	22
5.3	Ceramic Pattern	24
5.4	Pattern 1: Viewer	26
5.5	Pattern 2: Terminal	27
6	Conclusions & Future Work	29

Chapter 1

Introduction

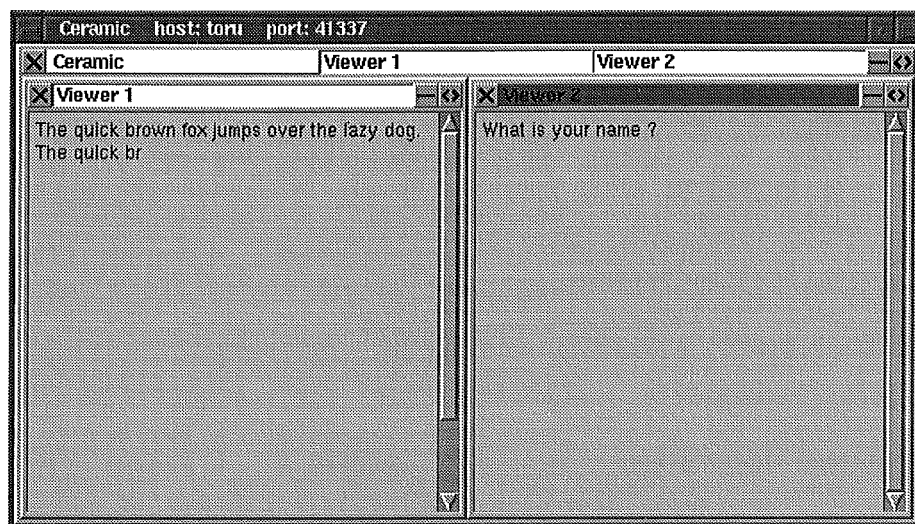


Figure 1.1: Ceramic Tiling Viewer screen capture.

Ceramic Tiling Viewer is a development tool which assists in observing and interacting with multi-thread programs. A sample screen capture is shown in Figure 1.1. Here Viewer 1, on the left, shows a thread outputting some text for the user to observe. The thread in Viewer 2, on the right, has asked a question and is waiting for the user to interact by inputting an answer. Waiting for input is indicated by Viewer 2's darker title bar.

Ceramic has been designed by capturing, reusing, and adapting design patterns from Mössenböck's Oberon0 viewer system [8]. Implemented in C++ Ceramic uses the Qt Toolkit (see Section 5.1.3) to assist in the graphical user interface development, and can operate under the Solaris and Linux platforms. Another feature are the hierarchical tiling viewers which are a hybrid of Elastic Windows developed by Kandogan & Shneiderman [7].

Sections 1.1–1.4 help clarify the problems associated with I/O for multi-thread pro-

grams that Ceramic solves. Section 1.5 outlines the structure for the rest of this report.

1.1 Context

A typical use for Ceramic will be for debugging multi-thread programs. There are situations where normal debugger programs cannot be used. For example consider embedded computer systems [2] running an embedded multi-thread microkernel such as OpenKernel [3]. For many embedded computer systems finding a compiler is difficult enough, let alone a debugger which will work cooperatively with the microkernel.

Another good use will be to observe what is going on in a multi-thread application. The observer can use the tiling viewer system to hone in on particular viewers with interesting output.

1.2 Problem

Observing and interacting with multi-thread programs can be difficult for the programmer. Writing output from multiple threads to one output device simultaneously can produce race conditions and jumbled output. Multiple threads simultaneously reading from the same device can also cause race conditions and input order mix-ups. Section 1.4 shows an example of this occurring.

1.3 Solution

A solution is to use windowing techniques. A window could be opened to control the each I/O stream of the multiple threads. This will eliminate: the race conditions, output being jumbled, and input order mix-ups.

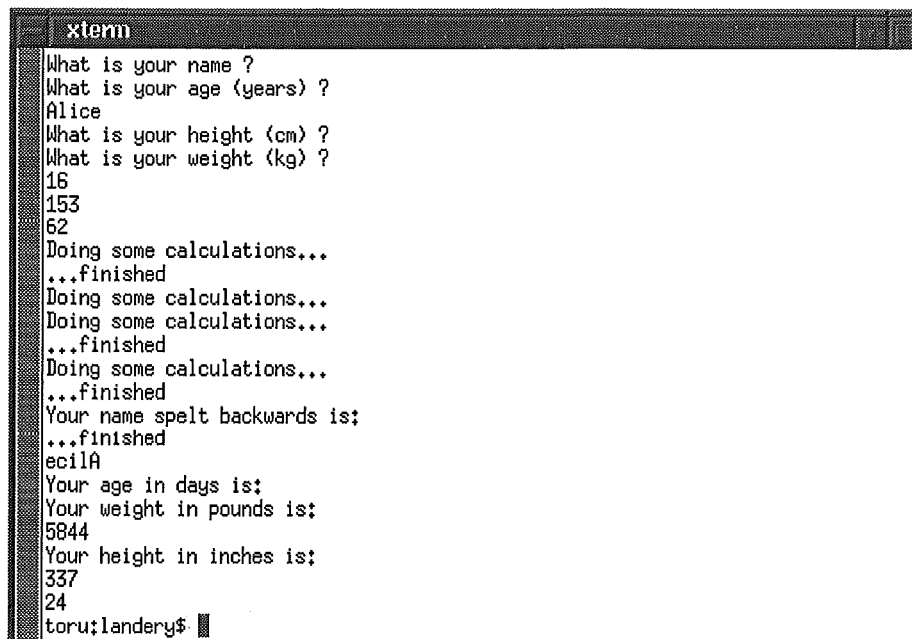
A multi-thread program may have several threads, furthermore threads may be created and destroyed during program execution. This produces a side-effect sub-problem of the user having to manage multiple windows.

Ceramic has been designed to eliminate the primary problem and relieve the side-effects produced by the sub-problem. Ceramic allows viewers (windows) to be opened to control I/O streams of multiple threads—solving the primary problem. The viewers are arranged in a hierarchical tiling layout to help relieve the side-effect of managing multiple viewers. The reasoning behind this will be explained in Chapter 5.

1.4 Example

To illustrate the problems of observing and interacting with multi-thread programs a 4-thread questionnaire program is used as an example. Each thread in the program asks a question, waits for the user to input an answer, does some calculations, and outputs a result. Figure 1.2 shows an execution of the program where Alice a fictitious 16 year, 153 cm, 62 kg girl has supplied the answers.

The output in Figure 1.2 looks jumbled up, making it hard to follow what is actually happening. More observant readers would have noticed that the results from the program were different to what Alice would expect. The program has calculated Alice to be 24 inches tall and weighing 337 pounds. The problem occurred because of a mix-up in the input order, 153 was read as Alice's weight and 62 was read as Alice's height.



```
xterm
What is your name ?
What is your age (years) ?
Alice
What is your height (cm) ?
16
What is your weight (kg) ?
153
62
Doing some calculations...
...finished
Doing some calculations...
Doing some calculations...
...finished
Doing some calculations...
...finished
Your name spelt backwards is:
...finished
ecilA
Your age in days is:
Your weight in pounds is:
5844
Your height in inches is:
337
24
toru:landery$
```

Figure 1.2: 4-thread questionnaire program I/O.

Figure 1.3 shows the same 4-thread questionnaire program example but this time it is using Ceramic for I/O.

Notice that the output in Figure 1.3 is much easier to comprehend. The mix-up in the input order could not occur this time. Hence the final results of the questionnaire are what Alice would expect.

1.5 Report Structure

The structure for the rest of this honours project report is as follows:

Chapter 2 discusses the background research and motivation for working on this project.

Chapter 3 outlines the project objectives.

Chapter 4 documents the design patterns captured from Oberon0.

Chapter 5 details the design and implementation of the Ceramic Tiling Viewer.

Chapter 6 draws some conclusions and indicates areas for future work.

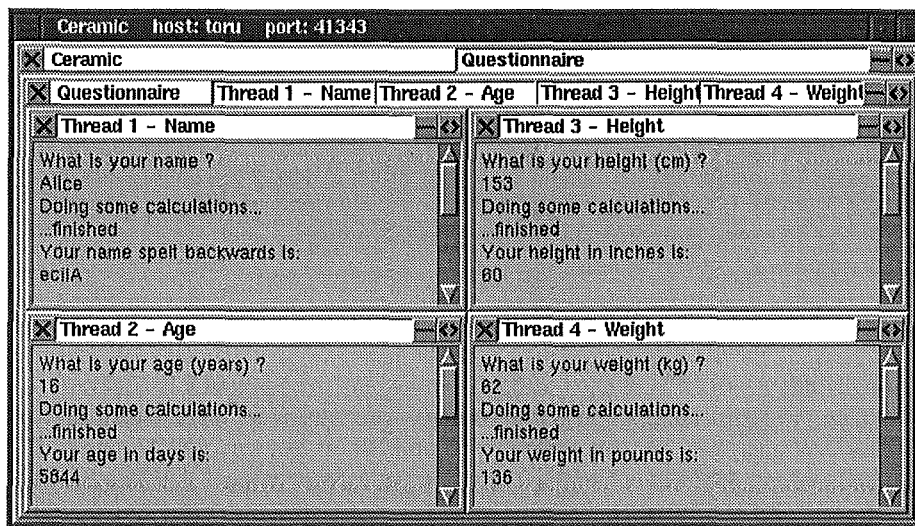


Figure 1.3: Questionnaire program using Ceramic for I/O.

Chapter 2

Background & Motivation

Before jumping into implementing a solution background research was undertaken. The topics searched were: tiling viewer systems, overlapping window systems, design patterns, embedded systems user interfaces, and multi-thread I/O tools. Searches for documents on embedded systems user interfaces and multi-thread I/O tools were not very successful.

The outline for the rest of this chapter is as follows: Section 2.1 discusses some previous work on tiling window systems, Section 2.2 introduces design patterns, and Section 2.3 discusses motivation for working on this project.

2.1 Tiling Window Systems

A *tiling* window system is defined as one in which any open window is always fully visible, i.e., windows are not allowed to overlap. The system attempts to manage the window locations, sizes, and side-effects to maximise the use of screen real estate while keeping window contents visible. Such a system typically determines the location and size of each window. When the location and/or size of a window changes, other windows are relocated and resized as needed. Since window size must decrease with an increase in the number of windows most systems limit the number of windows that can be opened simultaneously. They often provide facilities to iconify windows to release screen real estate for other windows.

In comparison an *overlapping* window system is defined as one in which the user manages a window's location and size in any way desired. Thus, the user controls the use of the screen real estate and the visibility of window contents. When the location and/or size of a window changes, other windows may be obscured, but their locations and sizes do not change. Because windows may overlap, a user can always choose to see a full screen's worth of contents of one window at the cost of completely obscuring all other windows.

The current trend towards the use of overlapping windows in windowing systems is based on the assumption that overlapping windows are clearly more beneficial to users than tiling ones. However there are situations where overlapping window systems are inferior to tiling window systems [1, 6].

The rest of this section describes some more relevant aspects of some existing tiling window systems with regard to this project.

2.1.1 Cedar

Cedar is a complete programming environment developed at Xerox PARC combining high-quality graphics, a sophisticated editor, and a variety of programming tools [10]. The component of most interest for this project is the Viewers window package which provides the basic display paradigm for Cedar.

The Viewers window package allow users to manipulate individual rectangular viewing areas called *viewers*. The term “viewer” corresponds to a “window” in many other systems. Viewers can present textual or graphical data to the user as well as receive keyboard and mouse input from the user.

Cedar uses a tiling strategy to place viewers on the screen. This is one of the most widely discussed aspects of the Cedar user interface, and often leads to heated, religious debates between its adherents and advocates of overlapping windows. The main part of the screen area is divided into two columns. The width and height of these columns can be easily adjusted by the user. Viewers within a column automatically share the space available by horizontally dividing up the screen space into equal portions. Opening a viewer allocates screen real estate to the viewer in the left column. The switch operation can move a viewer to the opposite column. Closing a viewer releases the space that the viewer currently occupies, and causes it to be displayed in iconic form at the bottom of the screen. A viewer can also be grown which allocates the full column to that viewer and closes all other viewers (to icons) in that column.

2.1.2 Oberon

In Project Oberon, Wirth & Gutknecht describe the design, development, and implementation of an operating system and compiler from scratch [11]. The quote from Einstein: “Make it as simple as possible, but not simpler” served as a signpost for the project approach. This resulted in a system of lucidity, efficiency, and compactness.

The Oberon operating system contains a display system which uses tiling viewers very similar to the Cedar environment. A tiling viewer system was chosen because of some inherent drawbacks in the more common overlapping window systems. They sighted three major drawbacks of overlapping systems. First, any efficient management of overlapping windows must use sophisticated clipping operations to draw partially covered windows. Secondly, there is a significant danger of covering windows completely and losing them forever. And thirdly, no canonical heuristic algorithms exist for automatic allocation of screen real estate to newly opened windows. They also stated that partial overlapping is desirable and beneficial in rare cases only, and it was hard to justify its additional complexity.

Like Cedar, Oberon divided the screen area into two columns, but each column had a different purpose. The left column, known as the user track, displays viewers from user programs, while the right column, known as the system track, displays viewers from system tools. When growing a viewer an overlay viewer was created which completely covered all other viewers in that track. The same viewer could be grown again creating a second overlay viewer which completely covered both tracks and therefore the entire screen display. The user had to close the overlay viewers to restore previous screen contexts. Oberon did not provide an iconic form for viewers causing the number of open viewers to be limited by screen real estate. Oberon’s automatic placement of new viewers heuristic strategy was different to Cedar. Instead of equally sharing screen space between viewers, Oberon splits the largest existing viewer in a given track into two halves of equal size, keeping all other viewers stable.

2.1.3 Oberon0

Oberon0 is a cut-down version of Oberon written by Mössenböck for students of object-oriented programming to study in detail [8]. Object-oriented programming is programming in the large and requires large, realistic examples. For this reason Mössenböck wrote Oberon0 as a large, realistic case study and wanted students to take time in studying it with pencil and paper in hand.

A large part of Oberon0 is written in conventional style. Not all data types are classes; not all operations are methods. This was a conscious design decision. Classes are used only where they make the program simpler or better extensible. Mössenböck wanted to show where classes make sense and where to do without them.

The functionality and implementation of Oberon0 is close to the Oberon. In fact to the procedures for the file system, mouse and screen control are borrowed from Oberon. For sake of simplicity, Oberon0 has only one column of viewers rather than two as in Oberon. Also viewers cannot be grown so there is no need for overlays. Otherwise the systems are externally identical. However internally Oberon0 is coded in Oberon-2 programming language while Oberon was coded in the Oberon programming language. Oberon0 implements most messages with methods and not with message records as in Oberon. Details from Oberon that would have inflated the source code without contributing to the object-oriented idea were omitted from Oberon0.

2.1.4 CUBRICON

CUBRICON Intelligent Window Manager (CIWM) is a knowledge-based system that automates windowing operations [4]. CIWM is a component of CUBRICON, a prototype knowledge-based multi-media human-computer interface. CIWM automatically performs window management functions on CUBRICON's screens. These functions are accomplished by the CIWM without direct human inputs, although the system provides for user override of the CIWM decisions.

The motivation for automated window management is based on the premise that, by freeing the user's cognitive and temporal resources from the task of managing the windows, more of these resources are available for the user's application domain activities. Some tasks in CUBRICON require a significant portion of the user's time spent on managing the window-based interface. The concept of automated window management offers great potential for enhancing human performance on these tasks.

CIWM combines tiling and overlapping window layout approaches to form a hybrid window configuration management methodology. CIWM uses a tiled windowing approach as a default, but allows the "tiled" windows to overlap adjacent windows when necessary. The system tries to get the "best of both worlds" by realizing the advantages of both types of windowing systems, while minimising the disadvantages.

The biggest fault with the system was the declutter operation which was invoked automatically when the screen became cluttered with too many windows. The declutter operation iconified windows based on their importance. Window importance was calculated from five factors: time of creation, contents, time since last interaction, frequency of interactions, and context. The problem was that windows in use were sometimes iconified. Users had problems with windows disappearing because the declutter algorithm was so complicated it seemed non-deterministic for the user.

2.1.5 Elastic Windows

Elastic Windows is a new approach to window management developed by Kandogan & Shneiderman [7]. It is based on three principles: hierarchical window organisation, multi-window operations, and space-filling tiled layout.

Hierarchical window organisation supports users structuring their work environment according to task. The group of windows associated with a task are organised as sub-windows of that task's window. Recursively each sub-window can also be organised as further sub-windows. This representation produces a hierarchical layout of windows. The hierarchical layout shows the semantic relationship between the contents of the windows by the spatial cues in their organisation.

Multiple window operations on groups of windows decreases the cognitive load on users by decreasing the number window operations. In Elastic Windows multiple operations are achieved by applying the operation to a group of windows at any level of the hierarchy. The results of the operations are propagated to windows inside that group recursively. This way groups of windows can be opened, closed, or resized with a single operation.

Space-filling tiled layout uses the screen space more productively, avoiding the wasted background of the overlapped windows approach. The tiled window layout maximises the visibility of windows for a task. People typically try to organise windows to be non-overlapping while working on a task, even when overlapping windows are allowed. Another factor was that overlapping window layouts are difficult to handle when a large number of windows must be visible at once, and they come and go rapidly.

Together the three principles of hierarchical organisation, multiple window operations, and space-filling tiled layout allows rapid task-switching, even when the number of windows is large.

2.2 Design Patterns

A design pattern is a description of communicating objects and classes that are customised to solve a general design problem in a particular context [5].

The design patterns captured in this report use the following template as a guideline for describing them in a uniform way:

Intent

A short statement describing what the design pattern does.

Aliases

Other well-known names for the pattern, if any.

Context

A typical context for the pattern's use.

Problem

Describes the particular design issue or problem the pattern addresses.

Forces

Lists the conflicts and constraints that the pattern resolves.

Solution

Details how the pattern solves the problem.

Structure

Diagram representation of the classes involved in the pattern using a notation based on the Object Modelling Technique [9].

Participants

Describes the classes and/or objects involved and their responsibilities within the pattern.

Collaborations

Shows how the participants collaborate to carry out their responsibilities.

Applicability

Situations where the pattern can be applied.

Known Uses

Examples of the pattern found in real systems.

2.3 Motivation

It is widely believed that overlapping windows are preferable to tiling ones [1]. A motivating factor for this project is to show a situation where a tiling window system is more beneficial for the user.

The main advantages of tiling window systems over overlapping window systems are as follows:

- Fewer window management operations for the user allows more cognitive resources to work on task related operations rather than to window management operations.
- All windows are fully visible so windows do not get obscured or lost.
- Uncluttered display appearance as windows are neatly arranged in tiled layout.
- Easier and faster to use. Experiments by Bly & Rosenberg suggested this was true for novice users and predicted it would also be true for expert users [1]. Hsu & Shen have showed that tiled windowing systems were superior to overlapping windows systems for high-demand tasks [6].

These are offset by the disadvantages of tiling window systems over overlapping window systems:

- Size and number of windows is restricted. All windows have to be visible, as number of windows increase their size must decrease.
- New windows often disrupt the display. Rearrangement of display required to incorporate a new window.
- Less control over display. Some window arrangements are not possible.

The following quote is from the Design Patterns book [5]:

Despite the book's size, the design patterns in it capture only a fraction of what an expert might know. It doesn't have any patterns dealing with concurrency or distributed programming or real-time programming. It doesn't have any application domain-specific patterns. It doesn't tell you how to build user interfaces, how to write device drivers, or how to use an object-oriented database. Each of these areas has its own patterns, and it would be worthwhile for someone to catalog those too.

The quote is motivation to find other design patterns not previously documented. Hopefully the patterns found will be of use to other designers.

Finally the main motivation is to solve the problem of observing and interacting with multi-thread programs. The background research did not find any such tools to solve this problem. This suggests that this is uncharted territory, giving more incentive to solve the problem using scientific techniques. Chapter 3 will now outline the project objectives.

Chapter 3

Project Objectives

The main objective of this project is to design and implement a semi-automated tiling viewer to observe and interact with multi-thread applications. The viewer should be able to be connected via a *virtual port* to receive and send data from separate multi-thread processes on different ports as shown in Figure 3.1.

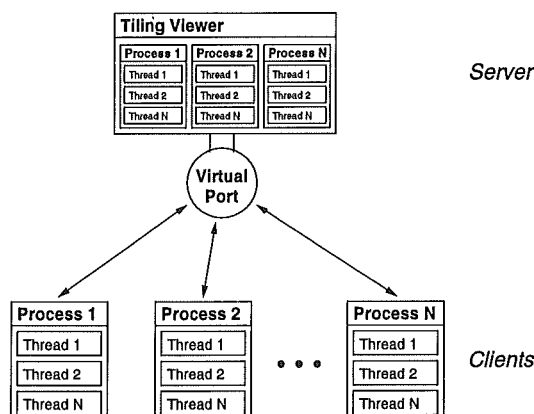


Figure 3.1: Tiling viewer server connecting clients via virtual port.

For example, the viewer should be able to display the output of an embedded application running on a target board linked to a Solaris station via a RS-232 port. A simple interface protocol for manipulating viewers, displaying data, and receiving data is required.

This project had the following objectives:

- Survey and evaluate functionality of existing tiling windowing systems and compare and capture some “potential” reusable design patterns.
- Reuse, adapt, and implement in C++ design patterns supporting a semi-automated tiling viewer on the Solaris platform. These patterns should be: simple, reliable, and have a straightforward interface. Portability and efficiency properties should be kept in perspective.
- Test the design patterns implementation for functionality, reliability, portability, and efficiency.

Chapter 4

Oberon0 Viewer System

As mentioned in Section 2.1.3, Oberon0 was written by Mössenböck as a large, realistic case study for students of object-oriented programming to study in detail. Oberon0 provided part of the functionality a multi-thread I/O tool with tiling viewers would require. For these reasons Oberon was subjected to a detailed study.

The study involved capturing the design patterns from the Oberon0 source code in the hope they would be reusable. A quick conversion of the Oberon0 source code from Oberon-2 to C++ was also done to help with familiarisation. This code was to be used as the basis for further development. The rest of this chapter documents the Oberon0 pattern that was captured.

4.1 Oberon0 Pattern

Intent

Present the user with a viewer system for input and output.

Context

The display system for the Oberon operating system.

Problem

The input and output of the viewer system must be controlled. Each command in the system must be able to read input from the user and send output to the screen. Management of the viewer interface and response to interactions from the user has to be provided.

Forces

This overall pattern resolves the following forces:

- Keeping the control of viewers separate from the contents displayed within them.
- Being extensible so different types of contents can be displayed in the viewers.

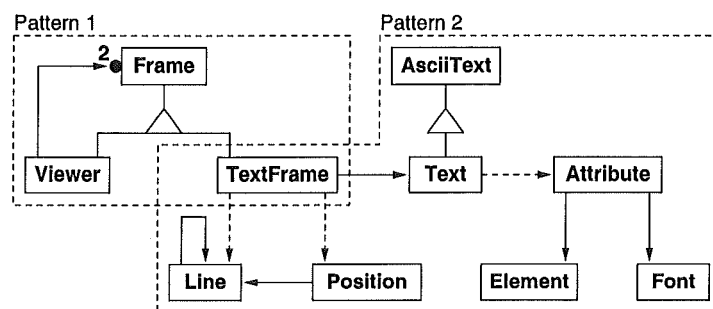
Solution

The solution is split into two patterns as shown in the following structure diagram. Each pattern has a different responsibility:

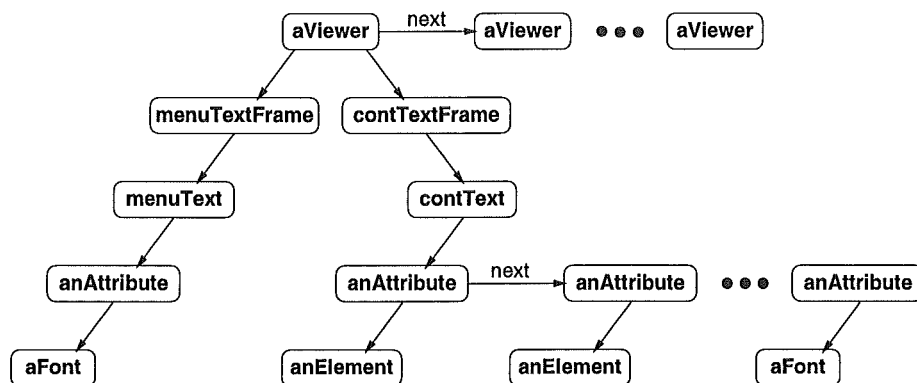
Pattern 1: Viewer provides an extensible viewer management system. Each viewer is capable of containing content which Pattern 2 details. The viewer controls the input and output from the content as well as managing size and position of itself.

Pattern 2: Text Editor is an example of the content for a viewer. Other types of content could be used, for example a graphics editor.

Structure



A typical Viewer pattern object structure might look like this:



Participants

Frame is an abstract class which providing an interface for displaying data and handling user input.

Viewer is responsible for directing on user input to its contents and manage the changes in size and position of itself.

AsciiText is responsible for maintaining a text buffer of ASCII characters.

Text is extends the capabilities of the ASCII text buffer to include support for fonts and the inclusion of pictures and other elements.

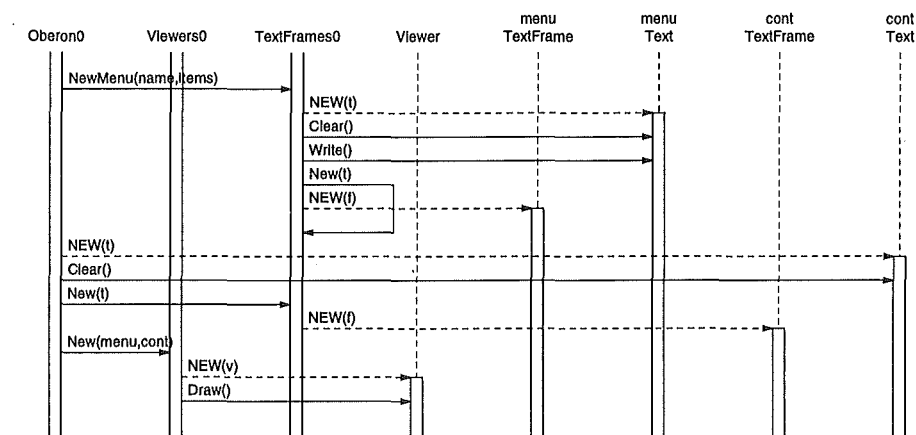
Element is an abstract class providing an interface so that a text can contain objects that are not ASCII characters.

TextFrame is responsible for displaying a text and handling user input.

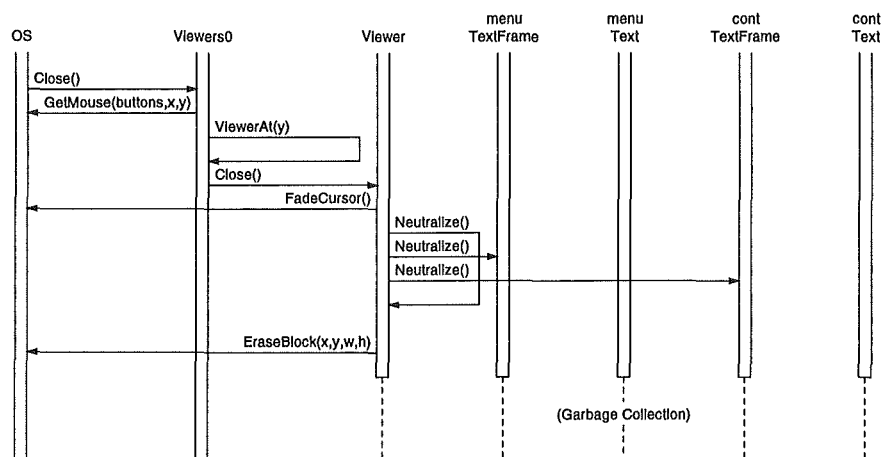
Collaborations

In the following interaction diagrams, OS, Oberon0, Viewers0, TextFrames0 are all modules within Oberon0. Each has been treated as an object to help illustrate the interactions.

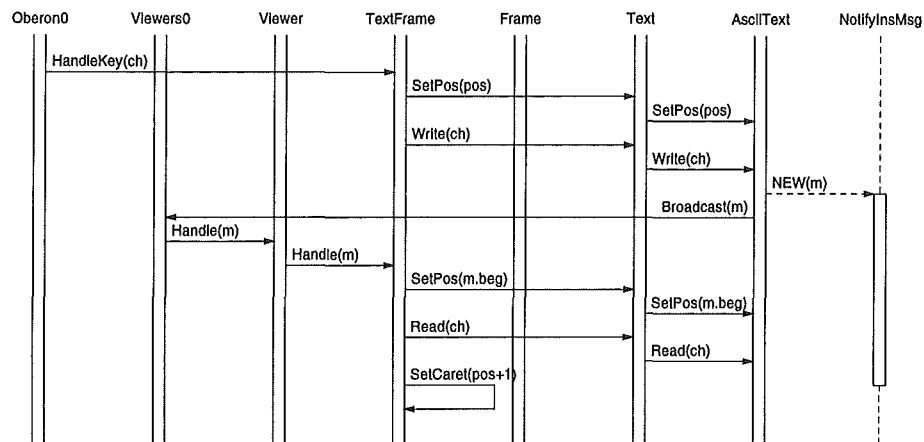
The following diagram shows the interactions between objects that occur when a Viewer is constructed.



The following diagram shows the interactions that occur between objects when a Viewer is closed. Notice that the objects are not explicitly deleted, this is because Oberon-2 has a garbage collection facility.



The following diagram shows the scenario when a character is inserted into a viewer content.



4.2 Pattern 1: Viewer

Intent

Provide an extensible viewer capable of handling screen display output and user input from the keyboard and mouse.

Aliases

This pattern is similar to the Composite pattern [5].

Context

The viewer system of the Oberon0 environment. Each viewer within the system handles a rectangular region of a screen display in which data can be viewed and edited. For the sake of simplicity, viewers in Oberon0 always contain exactly two frames: a menu frame with the name of the viewer and a list of Oberon commands, and a content frame to display the data. Each frame is responsible for displaying data (text, graphics, etc.) and handling user input (mouse clicks and keyboard input).

Problem

To be extensible viewers must be able to handle different types of data. For example one viewer could display text whilst another displays graphics. The viewers behaviour should not have to change to accommodate a different data type.

Forces

The Viewer pattern is useful when you have to balance the following forces:

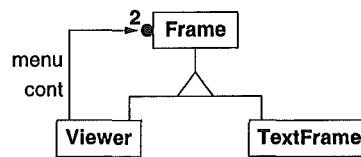
- The system must be extensible so that different types of data can be displayed in the viewers.
- Each viewer must be treated in a uniform way.

- Decouple the viewer from its contents.

Solution

Use an abstract frame class to provide an *interface* without completely implementing it. Due to this interface, a viewer knows which operations it can apply to the two frames (menu and content) it contains. And since a viewer can work with the abstract frame class, it can also work with classes inheriting the frame interface, such as text frames and graphics frames.

Structure



Participants

Frame is an abstract class which provides an interface without completely implementing it. A frame and its subclasses have two responsibilities:

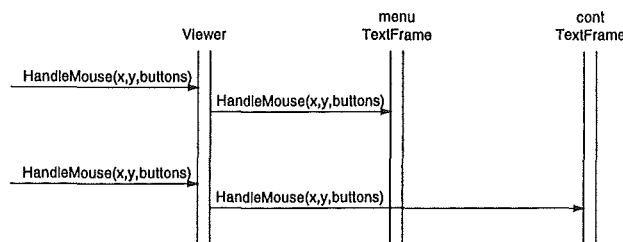
1. Display data (text, graphics, etc.).
2. Handle user input (mouse clicks and keyboard input).

Viewer is a subclass of a frame and therefore inherits the responsibilities of a frame. A viewer contains exactly two frames: a menu frame and a content frame. Part of the viewer's responsibility is to pass input not utilised by itself onto the menu frame and/or content frame. A viewer is also responsible for drawing its own border.

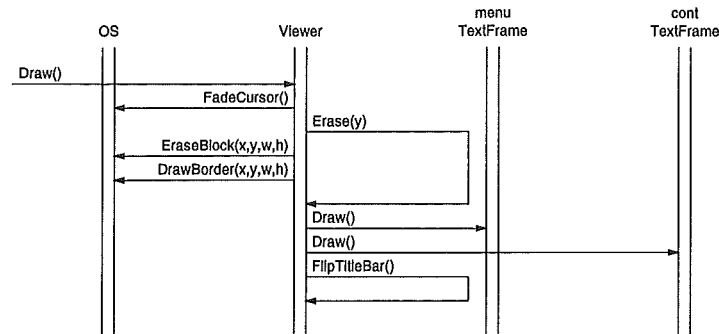
TextFrame is a concrete class derived from a frame for viewing and editing text. In this case both the menu frame and content frame of a viewer are **TextFrame** classes. This participant could be replaced with any other class that has been derived from a frame. For example a **GraphicFrame** could be used instead.

Collaborations

The following interaction diagram shows how the **Viewer** passes on the user input from the mouse. Internally the **Viewer** checks from the mouse y-coordinate to see which frame receives a **HandleMouse** message.



The following interaction diagram shows the scenario when the Viewer is drawn.



Known Uses

This pattern is similar to the Composite pattern but differs in two ways. Firstly the Viewer subclass is not an aggregator of the Frame class. In Oberon0 the menu frame and content frame objects are created by objects outside this pattern. Secondly the Viewer subclass has only two children: the menu frame and content frame. This was a design decision to keep the viewers programming simple.

4.3 Pattern 2: Text Editor

Intent

Provide a extensible text editor to permit displaying and editing of text within a viewer.

Context

The text editor is an example viewer content within the Oberon0 environment. The editor itself is independent of viewer but uses the abstract frame class as an interface so it will operate inside a viewer.

Problem

Text editors can be quite complicated. A realistic editor should support dynamic text buffer size, fonts, and allow pictures and other elements to be inserted in the text. The problem is structuring all of these features into a cohesive unit.

Forces

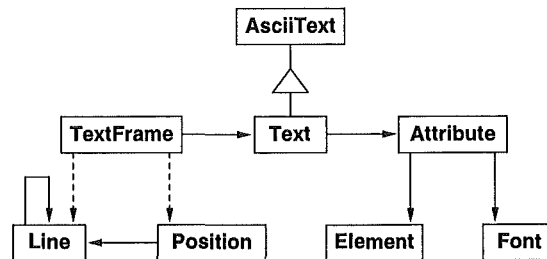
The Text Editor pattern is useful when you have to balance the following forces:

- Dynamic text buffer size.
- Ability to handle different attributes such as fonts and graphic elements.
- Has to operate within a viewer.

Solution

Divide responsibilities between the classes. **AsciiText** manages the text buffer. **Text** manages the text attributes which are fonts and elements. **TextFrame** handles the display of text within the viewer and processes user input from the keyboard and mouse.

Structure



Participants

AsciiText is responsible for maintaining a text buffer of ASCII characters and handling operations on the buffer such as: insertion, deletion, reading, writing, loading, and storing.

Text is a subclass of an ASCII text which extends the capabilities of the text buffer to include support for fonts and the inclusion of pictures and other elements.

Attribute is responsible for holding data about the type of objects within the text buffer. Data could be elements and/or a string of characters with a particular font.

Element is an abstract class providing an interface so that a text can contain objects that are not characters, without knowing the kinds of objects in advance. An element is responsible for drawing itself and handling mouse clicks.

Font holds the font characteristics association with a string of characters.

TextFrame is a subclass of frame and therefore inherits the responsibilities of a frame. More specifically a **TextFrame** is responsible for:

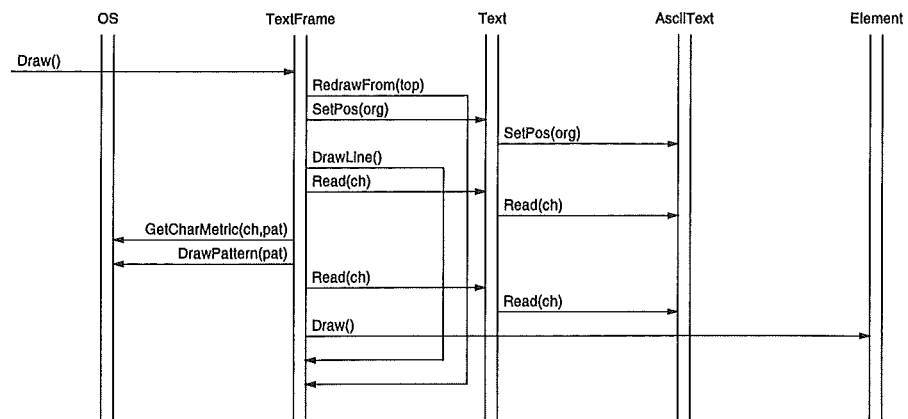
1. Display text.
2. Process keyboard input.
3. Process mouse clicks.

Position maps the screen display position of a line and the position in the text buffer.

Line maps the current line of text with the screen display.

Collaboration

The following interaction diagram shows the scenario when the Text Editor is asked to draw itself. This particular diagram shows the drawing of one character followed by one element.



Applicability

Use the Text Editor pattern when you need to create an extensible editor with the ability to handle fonts, elements, and other complex objects.

Chapter 5

Ceramic Tiling Viewer

Ceramic Tiling Viewer is a development tool which assists in observing and interacting with multi-thread programs. The name Ceramic came from ceramic tiles which have similar characteristics and properties Ceramic. Ceramic tiles often have decorative design patterns on them, they are reusable, robust, and flexible. Furthermore they are layed in a tiling fashion similar to the screen displays of Ceramic.

This chapter will discuss the design and implementation of the Ceramic. Section 5.1 gives some reasoning behind design choices. Section 5.2 explains the user interface of Ceramic. Finally Sections 5.3–5.5 document the design patterns in Ceramic in the style as Chapter 4.

5.1 Design Issues

Some design choices needed to be made while developing Ceramic. This section highlights some of the major decisions.

5.1.1 Hierarchical Tiling Viewers

Ceramic uses hierarchical tiling viewers similar to those found in Elastic Windows. The Elastic Windows system represents the latest work on tiling window systems. Once viewers were in a hierarchy multi-viewer operations could be performed. The layout of viewers uses a equal portion approach like Cedar. Each sub-viewer is given an equal of the screen real estate within a viewer. The algorithm which places viewers is automatic like the CUBRICON system but here the placement is deterministic by user standards.

5.1.2 Virtual Port

Currently the virtual port uses Unix TCP/IP sockets. The protocol between the client and server is a very simple one consisting of only four commands with the following purposes:

Open a viewer on the server, requires an arguments indicating the parent viewer in the hierarchy and the viewer's title.

Close a viewer on the server. Note this does not literally close the viewer but indicates to server that no more interaction will occur with this viewer. Literally closing

the viewer would cause a problem where the user could miss seeing that viewer's output.

Read a character from the server.

Write a string to the server.

5.1.3 Qt Toolkit

To assist in developing Ceramic's graphical user interface the Qt Toolkit was used. Several different toolkits were surveyed and Qt came up as the best for this project. Qt is an object-oriented C++ framework which dramatically cuts down on development time and complexity in writing user interface software. It uses a signal/slot mechanism provides true component programming. Reusable components can work together without any knowledge of each other, and in a type-safe way. Qt has been designed to be portable as well, versions exist for the Linux, Solaris, SunOS, FreeBSD, OSF/1, Irix, BSD/OS, NetBSD, SCO, and HP-UX platforms.

5.2 User Interface

There are two parts to the Ceramic user interface—the server and the client. The server is the interface for the user. It displays output and receives input from the user. Section 5.2.1 will explain this interface in more detail. The client is the interface for the programmer. Through the client the programmer can communicate with the user via the virtual port of the server. Section 5.2.2 will explain this interface in more detail.

5.2.1 Server

Figure 5.1 shows an example screen capture of Ceramic. Here a hierarchy of viewers has been opened to show the information from Minix (which is a small Unix operating system used for teaching about operating systems). Note that the screen real estate used by the borders and title bars of the viewers are over emphasised because the size of the window had to be small in order to make the text readable in this paper.

There is an error message in the fs viewer of Layer 3. To give more in screen real estate to Layer 3, iconify Layer 1, Layer 2, and Layer 3. This can be done in two ways, either select the iconify button on the right hand side of the viewer, or select the icon named "fs" in the title bar of the parent viewer. In either case this is an example of a multi-viewer operation. Figure 5.2 shows the result of this operation.

Also notice that the icon for Layer 1 has changed to green indicating some output has occurred. The icon for Layer 4 has change to red because it is waiting for some input. After clicking the icons the viewer(s) responsible for showing the indication will be highlighted. Note that input has a higher priority than output when colouring the icons. This is because a thread will become blocked on input.

These indicators are useful for informing the user of new information or requests for input. The user does not need to have a viewer open to check if something is happening.

5.2.2 Client

The following C header file, `cermic.h`, shows the functions the programmer can invoke to communicate with the user:

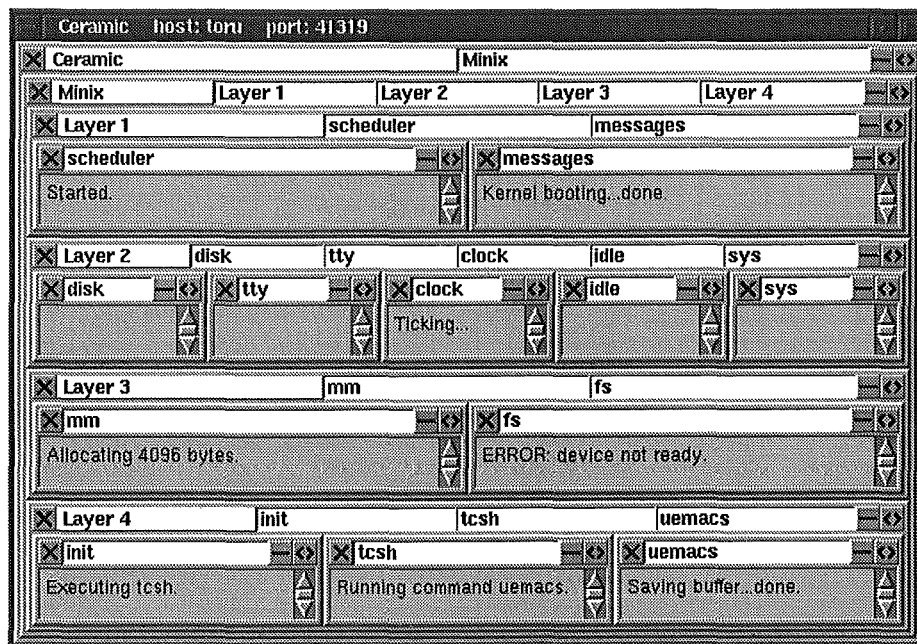


Figure 5.1: Ceramic handling lots of viewers.

```
/* ceramic.h - programmer interface */

typedef int VID;    /* viewer identification */

VID cinit(int argc, char **argv, const char *title);
VID copen(VID parent, const char *title);
int cclose(VID viewer);
int cprintf(VID viewer, const char *format, ...);
int cscanf(VID viewer, const char *format, ...);
int cgetc(VID viewer);
char *cgets(VID viewer, char *s, int n);
int cputc(VID viewer, int c);
int cputs(VID viewer, const char *s);
```

This client interface is very similar to `stdio.h`. Here is an example “Hello, World” program using the Ceramic programmer interface:

```
#include <ceramic.h>

VID viewer;

int main(int argc, char **argv)
{
    viewer = cinit(argc, argv, "Hello");
    cprintf(viewer, "Hello, World\n");
}
```

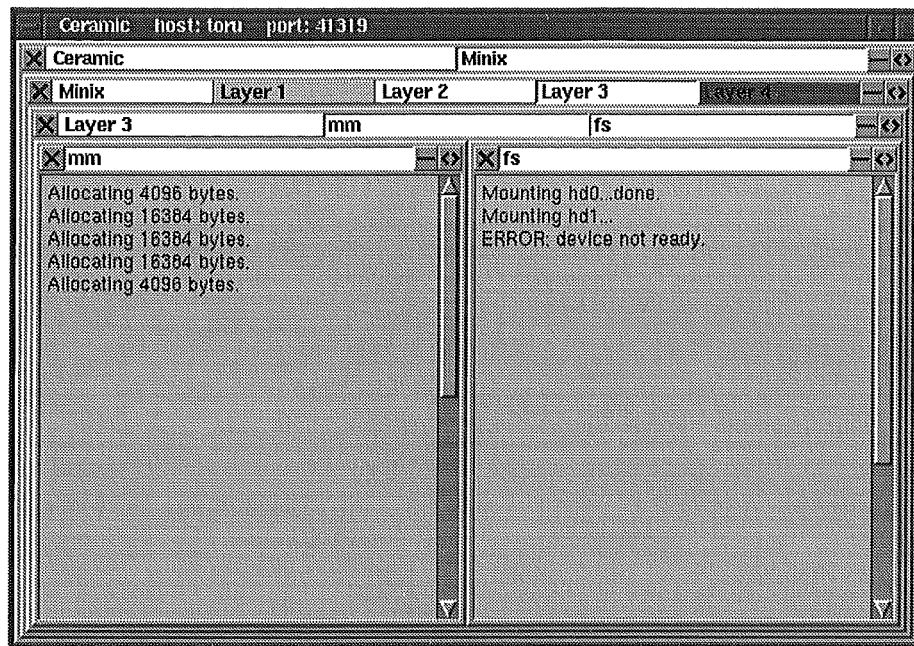


Figure 5.2: Ceramic revealing more information.

}

5.3 Ceramic Pattern

Intent

Present the user with a hierarchal viewer tiling system for input and output.

Context

The interface for a development tool which assists in observing and interacting with multi-thread programs.

Problem

The input and output of the viewer system must be controlled. Also the hierarchal tiling layout of the viewers needs to be managed.

Forces

This overall pattern resolves the following forces:

- Keeping the control of viewers separate from the contents displayed within them.
- Being extensible so different types of contents can be displayed in the viewers.

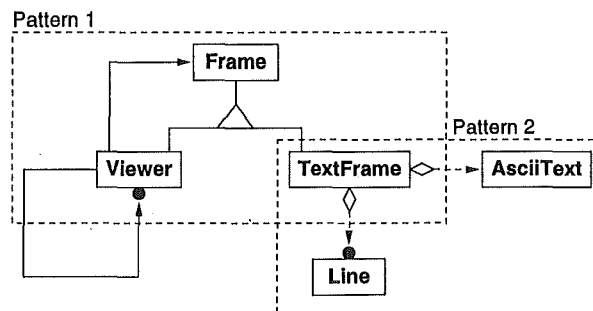
Solution

The solution is split into two patterns as shown in the following structure diagram. Each pattern has a different responsibility:

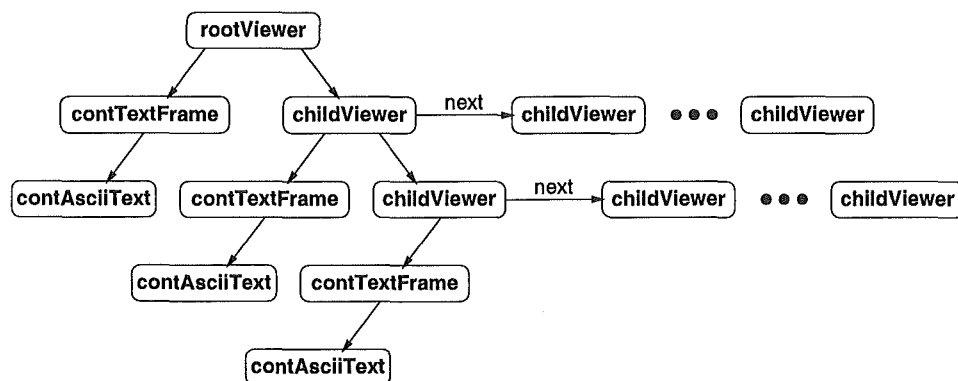
Pattern 1: Viewer provides an extensible viewer management system. Each viewer is capable of containing a content and multiple sub-viewers. The viewer controls the input and output from the content as well as managing size and position of its sub-viewers.

Pattern 2: Terminal is an example of the content for a viewer. For Ceramic its simple functionality of managing an ASCII text buffer was sufficient.

Structure



A typical Viewer pattern object structure might look like this:



Participants

Frame is an abstract class which providing an interface for displaying data and handling user input.

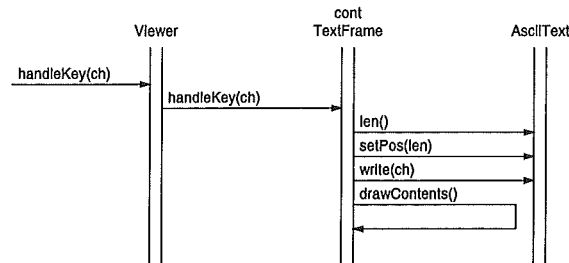
Viewer is responsible for directing on user input to its contents and manage the changes in size and position of its sub-viewers.

AsciiText is responsible for maintaining a text buffer of ASCII characters.

TextFrame is responsible for displaying a text and handling user input.

Collaborations

The following interaction diagram shows a scenario of how the objects interact to handle the insertion of a character into the content of a viewer.



5.4 Pattern 1: Viewer

Intent

Provide an extensible viewer capable of handling screen display output and user input from the keyboard and mouse.

Aliases

This pattern is similar to the Composite pattern [5].

Context

The viewer system of the Ceramic. Each viewer within the system is responsible for managing its content input and output, and sizing and positioning of its zero or more sub-viewers. This responsibility is recursive.

Problem

To be extensible viewers must be able to handle different types of data. For example one viewer could display text whilst another displays graphics. The viewers behaviour should not have to change to accommodate a different data type. The pattern must also be able to arrange the hierarchical tiling layout of the sub-viewers.

Forces

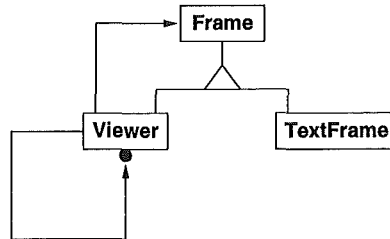
The Viewer pattern is useful when you have to balance the following forces:

- The system must be extensible so that different types of data can be displayed in the viewers.
- Each viewer must be treated in a uniform way.
- Decouple the viewer from its contents.
- Arrangement of viewers in a hierarchy.

Solution

Use an abstract frame class to provide an *interface* without completely implementing it. Due to this interface, a viewer knows which operations it can apply to the content. Since a viewer knows about itself it knows how to handle sub-viewers.

Structure



Participants

Frame is an abstract class which provides an interface without completely implementing it. A frame and its subclasses have two responsibilities:

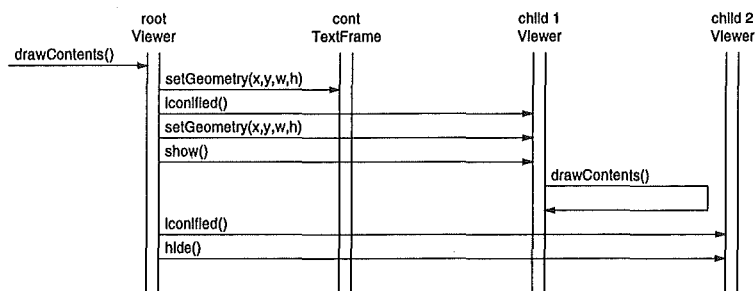
1. Display data (text, graphics, etc.).
2. Handle user input (mouse clicks and keyboard input).

Viewer contains a content frame and references to zero or more sub-viewers. The viewer is responsible for managing its content input and output, and sizing and positioning of its zero or more sub-viewers.

TextFrame is a concrete class derived from a frame for viewing and editing text.

Collaborations

The following interaction diagram shows the scenario when the **Viewer** is drawn. Here child 1 **Viewer** is to be shown on the screen but child 2 **Viewer** is iconified so it is hidden.



5.5 Pattern 2: Terminal

Intent

Provide a simple ASCII text buffer to permit displaying and editing of text within a viewer.

Context

The terminal is an example viewer content within Ceramic.

Problem

The terminal needs to maintain an ASCII text buffer and be able to display this in a viewer.

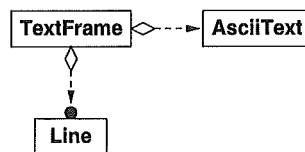
Forces

- Dynamic text buffer size.
- Has to operate within a viewer.

Solution

The solution borrows the `AsciiText` class from Oberon0 and uses it to maintain the text buffer. Once again a `TextFrame` class is used to allow the terminal to be used in a viewer.

Structure



Participants

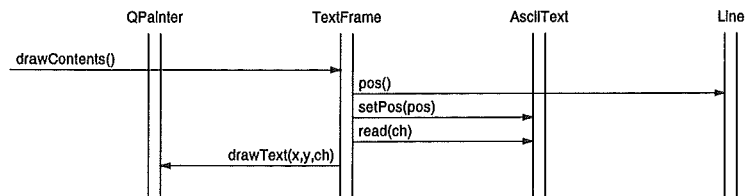
AsciiText is responsible for maintaining a text buffer of ASCII characters and handling reading and writing operations on the buffer.

TextFrame is responsible for handling input and output from the viewer and displaying the current contents of the ASCII text buffer.

Line maps the current line text with the screen display.

Collaborations

The following interaction diagram shows the scenario when a `TextFrame` draws a character on the screen. Note `QPainter` is a object from Qt to assist with painting on the screen.



Chapter 6

Conclusions & Future Work

The usage of design patterns gave a big insight into the Oberon0 system design. The pattern produced was not immediately relevant from the source code. After capturing these patterns Ceramic was born and the patterns were modified to accommodate the functionality of tiling viewers.

The project did not intend to advocate tiling viewers as the holy grail of window systems. More to show a situation where tiling viewers are a useful technique to solve a problem of managing multiple windows.

Ceramic is the beginnings of a useful development tool for programmers but there is still a lot more work to be done. Areas for future work include:

- Carry out a user evaluation study. Find out how users cope with the hierarchal tiling viewers.
- Investigate ways in which the output can better show the synchronisation of events.
- Improvement on the protocol for the virtual port.
- Implement clients for different programming languages.
- Extension to include embedded elements in the text frames. Provide common widgets that can be used in text frame.
- Develop other classes of contents frame. For example a graphics frame.

Bibliography

- [1] Sara A. Bly and Jarrett K. Rosenberg. A comparison of tiled and overlapping windows. In *Proceedings of ACM CHI'86 Conference on Human Factors in Computing Systems*, Windowing and Graphical Representation, pages 101–106, 1986.
- [2] A. Burns and A. Wellings. *Real-Time Systems and Their Programming Languages*. Addison-Wesley, Reading, MA, 1990.
- [3] M. de Champlain. Modèle réactif et réflexif pour méta machines à états finis. Ph.D. Thesis, École Polytechnique, Université de Montréal, Département de Génie Informatique, May 1995.
- [4] Douglas J. Funke, Jeannette G. Neal, and Rajendra D. Paul. An approach to intelligent automated window management. *International Journal of Man-Machine Studies*, 38(6):949–983, 1993.
- [5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
- [6] Shang H. Hsu and Chou Shen. Multiple monitors vs. windowing presentation styles for shop-floor controls. In *Proceedings of the Human Factors Society 36th Annual Meeting*, volume 1 of *COMPUTER SYSTEMS: Computer-Based Displays I*, pages 351–355, 1992.
- [7] Eser Kandogan and Ben Shneiderman. Elastic windows: Improved spatial layout and rapid multiple window operations. Technical Report CS-TR-3522, Human Computer Interaction Laboratory Center for Automation Research Institute for Systems Research Dept. of Computer Science, Univ. of Maryland, College Park, MD, September 1995.
- [8] Hanspeter Mössenböck. *Object-Oriented Programming in Oberon-2*. Springer-Verlag, New York, NY, 1993.
- [9] J. Rumbaugh et al. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1991.
- [10] Warren Teitelman. A tour through Cedar. *IEEE Software*, 1(2):44–73, April 1984.
- [11] N. Wirth and J. Gutknecht. *Project Oberon: The Design of an Operating System and Compiler*. Addison-Wesley, Reading, MA, 1992.